Міністерство освіти і науки України

Національний технічний університет України

“Київський політехнічний інститут імені Ігоря Сікорського”

Факультет інформатики та обчислювальної техніки

Кафедра інформатики та програмної інженерії

Комп’ютерний практикум №6

Технології паралельних обчислень

Тема: Розробка паралельного алгоритму множення матриць з використанням MPI-методів обміну повідомленнями «один-до-одного» та дослідження його ефективності

Виконав
студент групи ІП-11:
Панченко С. В.

Перевірила:
Стеценко І.В.

Київ 2024

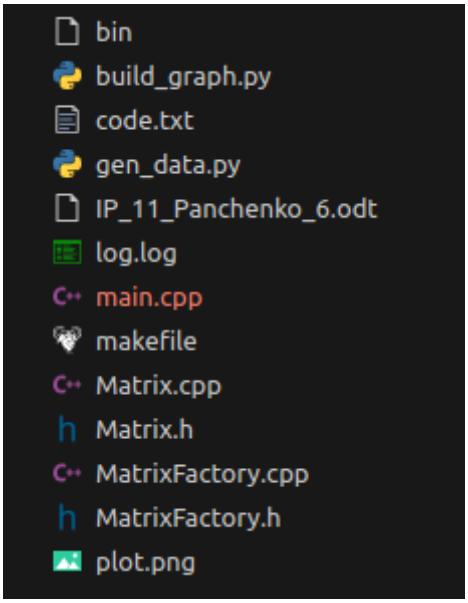
ЗМІСТ

1 ЗАВДАННЯ

1. Ознайомитись з методами блокуючого та неблокуючого обміну повідомленнями типу point-to-point (див. лекцію та документацію стандарту MPI).
2. Реалізувати алгоритм паралельного множення матриць з використанням розподілених обчислень в MPI з використанням методів блокуючого обміну повідомленнями (лістинг 1). 30 балів.
3. Реалізувати алгоритм паралельного множення матриць з використанням розподілених обчислень в MPI з використанням методів неблокуючого обміну повідомленнями. 30 балів.
4. Дослідити ефективність розподіленого обчислення алгоритму множення матриць при збільшенні розміру матриць та при збільшенні кількості вузлів, на яких здійснюється запуск програми. Порівняйте ефективність алгоритму при використанні блокуючих та неблокуючих методів обміну повідомленнями. 40 балів.

2 ВИКОНАННЯ

2.1 Структура проєкту



Bin — бінарний файл готової програми.

Makefile — файл для перетворення .h, .cpp — файлів в об’єктні файли, для їхнього лінкування із зовнішніми бібліотеками в один бінарний файл.

Matrix.h та Matrix.cpp — клас матриці, що використовує бібліотеку boost/serialization для серіалізації.

MatrixFactory.h та MatrixFactory.cpp — клас-фабрика, для створення матриць.

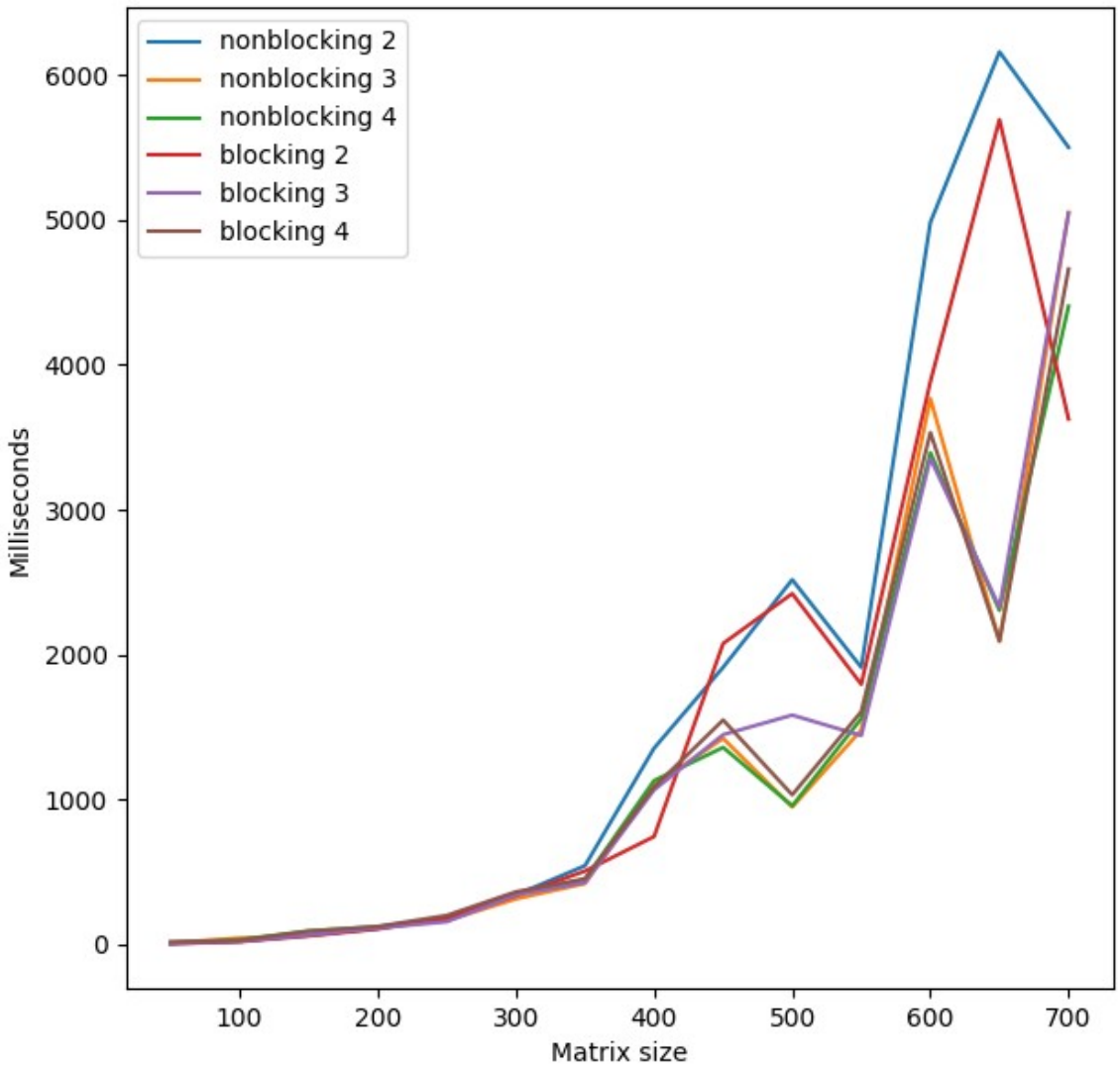
Main.cpp — ресурсний файл, що власне відповідає за роботу зі стандартом MPI, використовуючи реалізацію boost/mpi.

Log.log — файл, куди записуються результати прогону бінарного файлу з переданими при його запуску параметрами.

gen_data.py — скрипт на мові Python для повторного запуску бінарного файлу.

build_graph.py — скрипт на мові Python для побудови графіку на основі даних, записаних до log.log.

2.2 Результати



У результаті отримали такий графік, бачимо, що суттєва різниця у блокуючому та неблокуючому варіантах видно при 2 потоках та розмірі матриці в 700. Бачимо, що при збільшенні кількості потоків різниця поступово зменшується. Наразі можна сказати, що суттєвої різниці між двома варіантами немає у швидкодії, оскільки в алгоритмі все одно доводиться чекати на отримання надсилання та отримання матриць. Можливо, виграш при неблокуючому варіанті би за умови того, що потрібно виконувати більше операцій після надсилання, а хендлери до виконуваних тримати в буфері.

3 ВИСНОВОК

Під час лабораторної роботи опрацювали розробку паралельного алгоритму множення матриць з використанням MPI-методів обміну повідомленнями «один-до-одного» та дослідження його ефективності.

Побачили, що при стрічковому багатопотоковому множенні матриць суттєва різниця не спостерігається. Як було зазначено вище, то треба більше операцій, які можна виконувати незалежно, тоді різниця буде більш відчутною.

На мій погляд, код виглядає суттєво складніше та заплутаніше з використанням MPI-потоків, та додатково треба мати ще й спеціалізований компілятор, що ускладнює процес налагоджування програми.

ДОДАТОК А ТЕКСТИ ПРОГРАМНОГО КОДУ

Тексти програмного коду

(Найменування програми (документа))

Жорсткий диск

(Вид носія даних)

(Обсяг програми (документа), арк.)

Студента групи ІП-113 курсу

Панченка С. В

```
// ./Lab6/makefile
CC=mpiCC
CCRUN=mpirun
RUNFLAGS=-np 4
FLAGS=-std=c++20 -B /usr/include/boost/
BOOST_LIBS=-lboost_system -lboost_thread -lboost_mpi -lboost_serialization -lboost_log -lboost_log_setup -lpthread

BINARY=bin
CODEDIRS=.
CPPFILES=$(foreach D,$(CODEDIRS),$(wildcard $(D)/*.cpp))

all:$(BINARY)

$(BINARY):$(CPPFILES)
	$(CC) $(FLAGS) $(CPPFILES) $(BOOST_LIBS) -o $@

run:
	$(CCRUN) $(RUNFLAGS) $(BINARY)

clean:
	rm -rf $(BINARY) $(OBJECTS)

// ./Lab6/main.cpp

#define BOOST_LOG_DYN_LINK 1

#include <iostream>
#include <chrono>

#include <boost/mpi.hpp>
```

```
#include <boost/log/core.hpp>
#include <boost/log/trivial.hpp>
#include <boost/log/expressions.hpp>
#include <boost/log/sinks/text_file_backend.hpp>
#include <boost/log/utility/setup/file.hpp>
#include <boost/log/utility/setup/common_attributes.hpp>
#include <boost/log/sources/severity_logger.hpp>
#include <boost/log/sources/record_ostream.hpp>

#include "MatrixFactory.h"
#include "Matrix.h"

namespace mpi = boost::mpi;
namespace logging = boost::log;

static constexpr auto FROM_MAIN_THREAD_TAG = 1;
static constexpr auto FROM_TASK_THREAD_TAG = 2;
static constexpr auto MIN_VAL = 0;
static constexpr auto MAX_VAL = 1;

static const std::string NUMBER_OF_TASK_LE_ZERO = "Number of tasks is less or equal zero";
static const std::string MAIN_THREAD_TIME = "Main thread time: ";
static const std::string MILLIS = "ms";

static const std::string LOG_FILE = "log.log";
static const std::string LOG_FORMAT = "%Message%";
static constexpr auto LOG_OPEN_MODE = std::ios_base::app;

static const std::string BLOCKING = "blocking";

static constexpr char TAB = '\t';
```

```
void init();

int main(int argc, char* argv[]) {
    init();

    auto size = static_cast<unsigned>(std::stoi(argv[1]));
    auto blockType = std::string(argv[2]);
    auto isBlocking = blockType == BLOCKING;

    auto env = mpi::environment();
    auto world = mpi::communicator();

    auto tasksNum = world.size() - 1;
    if(tasksNum <= 0) {
        throw std::invalid_argument(NUMBER_OF_TASK_LE_ZERO);
    }

    auto isRowsLess = size < tasksNum;
    auto workingTasksNum = isRowsLess ? size : tasksNum;
    auto step = isRowsLess ? 1 : tasksNum;
    auto rank = world.rank();

    if(rank == 0) {
        auto factory = MatrixFactory();
        auto first = factory.GenerateMatrix(size, size, MIN_VAL,
MAX_VAL);
        auto second = factory.GenerateMatrix(size, size, MIN_VAL,
MAX_VAL);
        auto result = Matrix(size, size);

        auto start = std::chrono::high_resolution_clock::now();
```

```
                for(auto i = 0; i < workingTasksNum; ++i) {
                        if(isBlocking) {
                                world.send(i + 1, FROM_MAIN_THREAD_TAG, first);
                                world.send(i + 1, FROM_MAIN_THREAD_TAG,
second);
                        } else {
                                auto sent = std::vector<mpi::request> {
                                        world.isend(i + 1,
FROM_MAIN_THREAD_TAG, first),
                                        world.isend(i + 1,
FROM_MAIN_THREAD_TAG, second)
                                };
                                mpi::wait_all(sent.begin(), sent.end());
                        }

                }

                for(auto i = 0; i < workingTasksNum; ++i) {
                        auto tempResult = Matrix(size, size);
                        if(isBlocking) {
                                world.recv(i + 1, FROM_TASK_THREAD_TAG,
tempResult);
                        } else {
                                world.irecv(i + 1, FROM_TASK_THREAD_TAG,
tempResult).wait();
                        }

                        result.sum(tempResult);
                }

                // BOOST_LOG_TRIVIAL(info) << first;
```

```
                // BOOST_LOG_TRIVIAL(info) << second << LINE_FEED;
                // BOOST_LOG_TRIVIAL(info) << result << LINE_FEED;

                auto end = std::chrono::high_resolution_clock::now();
                auto millis = std::chrono::duration_cast<std::chrono::milliseconds>(end - start).count();

                BOOST_LOG_TRIVIAL(info) << blockType << TAB << size
                        << TAB << world.size() << TAB << millis;

            } else if(rank <= workingTasksNum) {
                auto first = Matrix();
                auto second = Matrix();
                auto result = Matrix(size, size);

                if(isBlocking) {
                    world.recv(0, FROM_MAIN_THREAD_TAG, first);
                    world.recv(0, FROM_MAIN_THREAD_TAG, second);
                } else {
                    auto received = std::vector<mpi::request> {
                        world.irecv(0, FROM_MAIN_THREAD_TAG, first),
                        world.irecv(0, FROM_MAIN_THREAD_TAG, second)
                    };
                    mpi::wait_all(received.begin(), received.end());
                }

                auto curRow = rank - 1;

            while(curRow < size) {
               for(auto j = 0; j < size; ++j) {
                  auto value = 0.0;
                  for(auto k = 0; k < size; ++k) {
```

```
                value += first[curRow][k] * second[k][j];
            }
            result[curRow][j] = value;

        }
        curRow += step;
    }

        if(isBlocking) {
            world.send(0, FROM_TASK_THREAD_TAG, result);
        } else {
            world.isend(0, FROM_TASK_THREAD_TAG, result).wait();
        }
    }

    return 0;
}

void init() {
    logging::add_file_log(
        LOG_FILE,
        logging::keywords::open_mode = LOG_OPEN_MODE,
        logging::keywords::format = LOG_FORMAT
    );
  logging::core::get()->set_filter(
    logging::trivial::severity >= logging::trivial::info
  );
}


// ./Lab6/MatrixFactory.h
```

```
#ifndef LAB6_MATRIXFACTORY_H
#define LAB6_MATRIXFACTORY_H

#include <vector>
#include <random>

class Matrix;

class MatrixFactory {
    public:
    MatrixFactory()=default;
    virtual ~MatrixFactory()=default;

    public:
    virtual Matrix GenerateMatrix(unsigned rows, unsigned cols,
                                  double minVal, double maxVal);

    protected:
    std::random_device rd;
    std::mt19937 rng = std::mt19937(rd());
    std::uniform_real_distribution<double> uni =
        std::uniform_real_distribution<double>(0, 1);
};


#endif //LAB6_MATRIXFACTORY_H


// ./Lab6/Matrix.cpp

#include "Matrix.h"
#include <iostream>
```

```
static constexpr char COMMA_SYMBOL = ',';
static constexpr char OPEN_BRACKETS_SYMBOL = '{';
static constexpr char CLOSING_BRACKETS_SYMBOL = '}';
static constexpr char SPACE_SYMBOL = ' ';

const std::string Matrix::MATRICES_NOT_THE_SAME_SIZE = "Matrices are not the same size";

Matrix::Matrix(unsigned int rows, unsigned int cols) {
	mat.resize(rows);
	for(auto& row : mat) {
		row.assign(cols, 0);
	}
}

std::vector<double>& Matrix::operator[](unsigned index) {
	return mat[index];
}

const std::vector<double>& Matrix::operator[](unsigned index) const {
	return mat[index];
}

const Matrix::InnerMat& Matrix::innerMat() const {
	return mat;
}

unsigned Matrix::rows() const {
	return mat.size();
}
```

```
unsigned Matrix::cols() const {
    return mat.front().size();
}

std::ostream& operator<<(std::ostream& out, const Matrix& matrix) {
    out << OPEN_BRACKETS_SYMBOL << SPACE_SYMBOL;
    for(auto i = 0u; i < matrix.rows(); ++i) {
        out << OPEN_BRACKETS_SYMBOL << SPACE_SYMBOL;
        for(auto j = 0u; j < matrix.cols(); ++j) {
            out << matrix[i][j] << SPACE_SYMBOL;
        }
        out << CLOSING_BRACKETS_SYMBOL << SPACE_SYMBOL;
    }
    out << CLOSING_BRACKETS_SYMBOL << SPACE_SYMBOL;
    return out;
}

void Matrix::sum(const Matrix& other) {
    if(rows() != other.rows() || cols() != other.cols()) {
        throw
std::invalid_argument(MATRICES_NOT_THE_SAME_SIZE);
    }

    const auto& otherInnerMat = other.innerMat();
    for(auto i = 0u; i < rows(); ++i){
        for(auto j = 0u; j < cols(); ++j) {
            mat[i][j] += otherInnerMat[i][j];
        }
    }
}
```

```
// ./Lab6/Matrix.h

#ifndef _MATRIX_H
#define _MATRIX_H

#include <vector>
#include <type_traits>
#include <boost/serialization/serialization.hpp>
#include <boost/serialization/vector.hpp>

class Matrix {
    public:
    Matrix()=default;
    Matrix(unsigned rows, unsigned cols);
    virtual ~Matrix()=default;

    public:
    using InnerMat = std::vector<std::vector<double>>;

    public:
    virtual const InnerMat& innerMat() const;
    virtual unsigned rows() const;
    virtual unsigned cols() const;

    public:
    virtual void sum(const Matrix& other);

    public:
    friend std::ostream& operator<<(std::ostream& out, const Matrix&
matrix);
    virtual std::vector<double>& operator[](unsigned index);
    virtual const std::vector<double>& operator[](unsigned index) const;
```

```
        private:
        friend class boost::serialization::access;
        template<class Archive>
                void serialize(Archive& ar, const unsigned int version) {
                        ar & mat;
                }

        protected:
        InnerMat mat;

        protected:
        static const std::string MATRICES_NOT_THE_SAME_SIZE;
};


#endif //LAB6_MATRIX_H


// ./Lab6/MatrixFactory.cpp

#include "MatrixFactory.h"
#include "Matrix.h"

Matrix MatrixFactory::GenerateMatrix(unsigned rows, unsigned cols, double minVal, double maxVal) {
        auto matrix = Matrix(rows, cols);
        for(auto i = 0u; i < matrix.rows();++i) {
                for(auto& el : matrix[i]) {
                        el = (maxVal - minVal) * uni(rng) + minVal;
                }
        }
```

```
        return matrix;
}
```